Automatic Abstraction for Synthesis and Verification of Deterministic Timed Systems*

Hao Zheng
Electrical Engineering Dept.
University of Utah
hao@vlsigroup.elen.utah.edu

Chris Myers
Electrical Engineering Dept.
University of Utah
myers@vlsigroup.elen.utah.edu

ABSTRACT

This paper presents a new approach for synthesis and verification of asynchronous circuits by using abstraction. It attacks the state explosion problem by avoiding the generation of a flat state space for the whole design. Instead, it breaks the design into sub-blocks and conducts synthesis and verification on each of them. Using this approach, the speed of synthesis and verification improves dramatically. This paper introduces how abstraction is applied to timed Petri-nets to speed up synthesis and verification.

1. INTRODUCTION

In order to continue to produce circuits of increasing speed, designers are considering aggressive circuit design styles such as self-resetting or delayed-reset domino circuits. These design styles can achieve a significant improvement in circuit speed as demonstrated by their use in a gigahertz research microprocessor (guTS) at IBM [?]. Designers are also considering asynchronous circuits due to their potential for higher performance and lower power as demonstrated by the RAPPID instruction length decoder designed at Intel [22]. This design was 3 times faster while using only half the power of the synchronous design. The correctness of these new *timed circuit* styles is highly dependent upon their timing. Extensive verification is also necessary during the design process. Unfortunately, these new circuit styles cannot be efficiently and accurately synthesized, analyzed, and verified using traditional static timing analysis methods. This lack of efficient analysis tools is one of the reasons for the lack of mainstream acceptance of these design styles.

The synthesis and verification of timed circuits and especially timed asynchronous circuits often requires state space exploration which can explode even for modest size examples. In [13], a direct synthesis method is proposed which synthesizes timed circuits directly from signal transition graphs without state exploration, but a similar approach cannot easily be applied to verification. To reduce the complexity incurred by state exploration, abstraction is necessary. In [1, 2, 21], safe approximations of internal signal behavior are presented to reduce the state space under consideration, but these methods suffer exponential complexity in the number of memory elements. In VIS [7], non-determinism is used to abstract the behavior of some circuit signals. It is often too conservative, and can introduce unreachable states which may exhibit hazards. In [20], a model checker is proposed based on hierarchical reactive machines. By taking advantage of the hierarchy information, it only tracks active variables so that the state space is reduced and verification time is improved. This approach, however, is best suited for software which has a more sequential nature. In [9], an abstraction technique is proposed for validation coverage analysis and automatic test generation. It removes all datapath elements which do not affect the control flow and groups the equivalent transitions together, thus resulting in a dramatic reduction in the state space. It is difficult, however, to distinguish the control from the datapath without help from the designers. In [14], an abstraction approach for the design of speed-independent asynchronous circuits from change diagrams is described. In this approach, each subcircuit is designed individually, and they are then recombined to produce the final circuit. This approach, however, does not address timing issues. [11] presents a divide-and-conquer method for synthesis of asynchronous circuits. This method breaks the state graph for a given problem into a number of simpler modular subgraphs for each output. Each modular subgraph is solved individually. The result of these small subgraphs are then integrated together contributing to the solution of the given problem. Although this makes synthesis and verification easier, the quality of the final solution may depend on the order in which the outputs are processed. Also, this method generates a complete state graph before it breaks the state graph, which is highly undesirable for large complex designs. In [5], Belluomini described the verification of domino circuits using `ATACS`. She found out that verifying flat circuits even of a moderate size is too difficult to be done by `ATACS`, but with some hand abstraction, the verification is completed quickly. Although doing abstraction by hand is possible, it requires an expert user and methods must be developed to check that the abstractions are a reliable model of the underlying behavior. This is the major motivation of this work.

Our approach begins with a high-level language, such as VHDL, which models a system hierarchically, and our method

*This research is supported by NSF CAREER award MIP-9625014, SRC contract 99-TJ-694, and a grant from Intel Corporation.

TAU'00, December 4-5, 2000, Austin, Texas.

utilizes the hierarchy information to compile each individual component into a graphical representation for synthesis and verification. This paper proposes an abstraction technique applied to *timed Petri-nets*. This approach partitions the design into blocks of manageable size. Then, a block is chosen as a target of synthesis or verification, and the rest of the blocks and the environment are merged for the target. By taking advantage of the hierarchical information, the environment for the block is simplified, and the state space of the target block is reduced resulting in a substantial savings in synthesis and verification time.

2. TIMED PETRI-NETS

This section briefly introduces timed Petri-nets (TPN), the graphical model to which our high-level specification language is compiled [19]. A one-safe TPN is modeled by the tuple $\langle P, T, F, M_0, \Delta \rangle$ where P is the set of places, T is the set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, $M_0 \subseteq P$ is the initial marking, and Δ is an assignment of timing constraints to places. There are three kinds of transitions: $s+$ changes signal s from 0 to 1, and $s-$ changes s from 1 to 0, and \$ which is a *sequencing transition*. A *marking* is a subset of places. For a place $p \in P$, the *preset* of p (denoted $\bullet p$) is the set of transitions connected to p (i.e., $\bullet p = \{t \in T \mid (t,p) \in F\}$), and the *postset* of p (denoted $p\bullet$) is the set of transitions to which p is connected (i.e., $p\bullet = \{t \in T \mid (p,t) \in F\}$). Presets and postsets for transitions are similarly defined. A timing constraint consisting of a lower and upper bound is associated with each place in the TPN (i.e., $\Delta(p_i) = \langle l_i, u_i \rangle$). The lower bound is a nonnegative integer while the upper bound is an integer greater than or equal to the lower bound or ∞. A *marked graph* is a TPN in which every place has at most one transition in its preset and postset. A *choice place* is one in which there are multiple transitions in its postset (i.e., $\mid p\bullet \mid > 1$).

A transition t is enabled in a marking M if all of the places in t's preset are in the marking (i.e., $\bullet t \subseteq M$). A clock c_i is associated with each place p_i in a marking. The clock is initialized to zero when the place is put into the marking. All clocks in a marking increase uniformly. A clock is *satisfied* when it reaches the lower bound on the corresponding place (i.e., $c_i \geq l_i$). A clock is *expired* when it reaches the upper bound on the corresponding place (i.e., $c_i \geq u_i$). A transition cannot occur until it is enabled and all clocks on places in its preset are satisfied. A transition must occur when all clocks on places in its preset are expired. The result of a transition firing is that all places in its preset are removed from the marking and the corresponding clocks are discarded. Next, all places in its postset are added to the marking and the corresponding clocks are introduced and initialized to zero (i.e., $M = M - \bullet t + t\bullet$).

3. FUNDAMENTALS OF TRACE THEORY

This section provides a brief overview of trace theory. Trace theory was first applied to the verification of the speed-independent circuits by Dill [10]. Later, timing was added so that trace theory can be applied to the verification of timed circuits [8, 26]. We show some useful properties and lemmas which are used in the later proofs in this paper.

A *timed trace*, x, for a circuit is a finite or infinite sequence of events (i.e., $x = e_0 e_1 \ldots$). Each timed event is of the form $e_i = (w_i, t_i)$ where w is a wire name in the circuit, which represents a logic value change on that wire, and t is a rational number indicating when that change happens. A timed trace must also satisfy the following two properties:

- *Monotonicity*: $t_i \leq t_{i+1}$ for all $i \geq 0$, and
- *Progress*: if x is infinite then for any time t there exists an i such that $t_i > t$.

Given a trace $x = e_1 e_2 \ldots$ and a set of signals D, the function $\mathbf{del}(D)(x)$ is defined as follows:

$$\mathbf{del}(D)(x) = e_1 y \qquad \text{if } w_1 \notin D \tag{1}$$

$$\mathbf{del}(D)(e) = y \qquad \text{if } w_1 \in D \tag{2}$$

where $y = \mathbf{del}(D)(e_2 e_3 \ldots)$ and $e_1 = (w_1, t_1)$. This function deletes all events of a trace whose wire names are in D. It is extended naturally to sets of traces. Given a set of traces X, the function inverse delete $\mathbf{del}^{-1}(D)(X)$ is the set $\{x' | \mathbf{del}(D)(x') \in X\}$. This function returns the set of traces which would be in X if all events with wire names in D are deleted. Intuitively, if x is a trace not containing symbols from D, $\mathbf{del}^{-1}(D)(x)$ is the set of all traces that can be generated by inserting events in D at any time into x. Some useful properties of these two functions are below:

$$\mathbf{del}(D)(X) = \emptyset \Leftrightarrow X = \emptyset \tag{3}$$

$$\mathbf{del}(D)(\mathbf{del}^{-1}(D')(X)) = \mathbf{del}^{-1}(D')(\mathbf{del}(D)(X)) \quad \text{when } D \cap D' = \emptyset \tag{4}$$

$$\mathbf{del}(D)(\mathbf{del}^{-1}(D)(X)) = X \tag{5}$$

$$\mathbf{del}(D)(X \cap X') \subseteq \mathbf{del}(D)(X) \cap \mathbf{del}(D)(X') \tag{6}$$

A *prefix-closed trace structure* T is a four-tuple $\langle I, O, S, F \rangle$. I is a set of input wires, and O is a set of output wires where $I \cap O = \emptyset$. $A = I \cup O$ is the *alphabet* of the structure. S is the *success set* which contains all successful traces of a system. F is the *failure set* which contains all failure traces of a system. $P = S \cup F$ is the set of all possible traces of a system. A trace structure must be *receptive*, meaning that $PI \subseteq P$. Intuitively, this means a circuit cannot prevent the environment from sending an input.

Composition ($\|$) combines two circuits into a single circuit. Composition of two trace structures $T = \langle I, O, S, F \rangle$ and $T' = \langle I', O', S', F' \rangle$ is defined when $O \cap O' = \emptyset$. To compose two trace structures, the alphabets of both trace structures must first be made the same by adding new inputs as necessary to each structure. Inverse delete is extended to structures for this step as follows:

$$\mathbf{del}^{-1}(D)(T) = \langle I \cup D, O, \mathbf{del}^{-1}(D)(S), \mathbf{del}^{-1}(D)(F) \rangle \tag{7}$$

This is defined only when $D \cap A = \emptyset$.

After the two alphabets of the two structures are made to match, we need to find the traces that are consistent with the two structures. The intersection of these two trace structures is defined as follows:

$$T \cap T' = \langle I \cap I', O \cup O', S \cap S', (F \cap P') \cup (P \cap F') \rangle \tag{8}$$

This is defined only when $A = A'$ and $O \cap O' = \emptyset$. From this definition, a success trace in the composite must be a success trace in both components. A failure trace in the composite is a possible trace that is a failure trace in either component. The possible traces for the composite is $P \cap P'$.

Composition can now be defined as follows:

$$T \parallel T' = \mathbf{del}^{-1}(A' - A)(T) \cap \mathbf{del}^{-1}(A - A')(T') \quad (9)$$

Another useful operation is **hide** which is used to make some wires *internal* to the circuit. Given a trace structure T, $\mathbf{hide}(D)(T)$ is defined as follow:

$$\mathbf{hide}(D)(T) = \langle I, O - D, \mathbf{del}(D)(S), \mathbf{del}(D)(F) \rangle \quad (10)$$

where D is the set of wires to be hidden.

A trace structure is *failure-free* if its failure set is empty. Given two trace structures, T and T', we say T *conforms* to T' (denoted $T \preceq T'$) if $I = I', O = O'$, and for *all* environments E, if $E \parallel T'$ is failure-free, so is $E \parallel T$. Intuitively, if a system using T' cannot fail, neither can a system using T.

The following lemma gives a simple sufficient condition to determine conformance between two trace structures.

LEMMA 1. *$T \preceq T'$ if $I = I', O = O', F \subseteq F'$, and $P \subseteq P'$.*

The condition $F \subseteq F'$ assures that if the environment does not cause a failure in T', it does not cause a failure in T. The condition $P \subseteq P'$ assures that if T' does not cause a failure in the environment, T does not cause one.

The next lemma shows that if T conforms T', this conformance is maintained in any environment.

LEMMA 2. *If $T \preceq T'$ and T'' is any trace structure, then $T \parallel T'' \preceq T' \parallel T''$.*

Proofs of these lemmas can be found in [10].

4. AUTOMATIC ABSTRACTION

Synthesis and verification of timed systems are based on complete state space exploration. The state space can be derived by exhaustively firing all possible transition sequences in the system. The number of states grows exponentially as the complexity of the design grows. Therefore, synthesis and verification of large and complex systems is difficult or even impossible because of state explosion. In general, a large and complex design consists of multiple subsystems (blocks) which are connected by signals. To synthesize or verify a timed system, an environment describing the input behavior of the system needs to be provided. This environment is referred to as *system* environment. Each block either connects to other blocks, the system environment, or both. Since the size and complexity of each block is often much less than the whole system, we would like to synthesize or verify each block individually to reduce the total time taken to finish the process for all blocks. After the results for all blocks are available, they are integrated together to form the solution of the whole system. If a block is chosen as the target of synthesis or verification, the rest of the blocks and the system environment together form the environment for the target block, which is referred to as the *block environment*. Although the state space of the block is reduced, the state space of the block environment is increased, and the total state space remains unchanged. Before synthesis or verification starts, the block environment needs to be simplified. This is where abstraction comes into play.

In general, the block environment for a block contains two groups of signals: interface signals connecting the block environment to the block; and internal signals which may include interface signals or internal signals of the system. To synthesize or verify a block, however, only the behavior of the interface signals is of concern. If the information of the internal signals of the block environment can be canceled while the interface behavior is kept in a conservative way, the block environment can be simplified and the total state space can be reduced. Therefore, the basic idea behind abstraction is to choose a block with manageable size and complexity, group the rest of the blocks and the system environment as the block environment for the target block, identify all internal signals of the block environment by using the specified structural information, and remove all information related to them in such a way that the essential behavior of the interface signals is maintained.

We have incorporated the above idea into the synthesis and verification tool ATACS. This tool reads a high-level specification in a language such as VHDL and translates it to a TEL structure [4, 27]. To simplify the description in this paper, we describe the approach taken on a more common data structure, TPNs. To apply abstraction to TPNs, first all transitions on internal signals of the block environment are abstracted to sequencing transitions; then a safe transformation procedure is called to remove sequencing transitions and the related places from the TPN, when possible.

The following figures show how abstraction works. Figure 1 is the block diagram of a 2-stage FIFO. Figure 2 is the TPN of the 2-stage FIFO before abstraction. In order to simplify the diagram, we have removed the places between the transitions. The bullets on arcs between two transitions indicate that the place between these transitions is initially marked. If we want to consider fifo0 only, the left and right environments and fifo1 form the block environment for fifo0, and signals req(2) and ack(2) are now internal signals of the block environment. Abstraction changes all transitions on those signals into sequencing transitions. Figure 3 shows the TPN for fifo0 after abstraction.

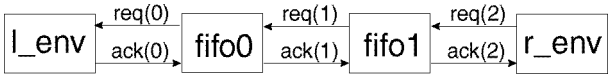

Figure 1: Block Diagram of a 2-stage FIFO.

5. SAFE TRANSFORMATIONS

The last section describes how the internal signals of the environment are identified. This section describes the procedures to remove those internal signals. Some techniques have been presented to simplify untimed Petri nets. Suzuhi and Murata [23, 24] present a method of stepwise refinement of transitions and places into subnets. They show a sufficient condition that such subnets must satisfy, which are dependent on the structure and initial marking of the net. The resulting net has the same liveness and safety properties as that of the original net. However, this refinement process has to be repeated every time the initial marking is changed. This makes automating the refinement difficult. Berthelot [6] presented several transformations that depend only on the structure of the net. In [18, 12, 17], several transformations for marked graphs are presented. These transformations cut places and transitions in the graph while preserving liveness and safety. These transformations, however, are only applied to untimed Petri nets.

As described in the last section, internal signals of the en-

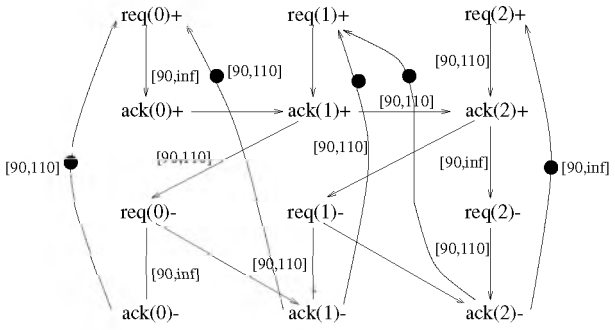

Figure 2: TPN of 2-stage FIFO before abstraction.

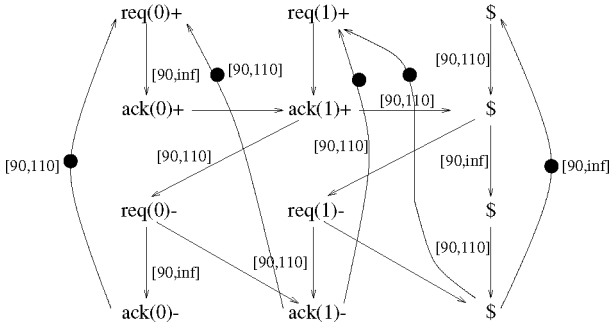

Figure 3: TPN of 2-stage FIFO after abstraction.

vironment can be removed during synthesis and verification as long as the interface behavior of the environment with the internal signals are preserved. Abstraction converts internal signals of the environment into sequencing transitions. *Safe transformations* remove sequencing transitions from TPNs under certain conditions. First, removal of a sequencing transition should never change the untimed semantics of the environment. The environment accepts signal transitions from the circuit and responds sometime later with transitions on its signal wires. The environment's signals either depend on the circuit's signals directly, or depend on some internal signals which then depend on circuit signals. After transformations, it is required that the environment still generates the same signal transitions in response to the signal transitions from the circuit. After transformations, all environment signal transitions depend directly on the circuit signals. Second, the timing information of the signal transitions produced by the environment must be preserved in a conservative fashion. When the environment sees transitions on the circuit's signals, it fires transitions on its signals after some delay. The firing sequence depends on the relative timing bounds of the environment and circuit signals. If these relative timing bounds are preserved conservatively after transformations, the environment generates all the same traces as before transformations. It may, however, also produce some new traces since timing is only preserved conservatively. We call the environment after the abstraction and safe transformations the *abstracted* environment. The circuit synthesized from the block with the abstracted environment accepts a superset of inputs of the unabstracted environment. The synthesized circuit may be redundant compared with the one synthesized with the unabstracted environment. By constraining the behavior of the environment, the circuit behaves in the same way as specified. For verification, the result might be a false negative, but never a false positive.

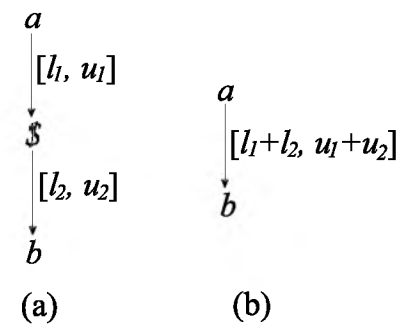

Figure 4: Safe transformation 1.

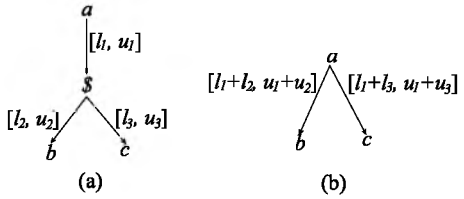

Figure 5: Safe transformation 2.

The correctness of safe transformations can also be described by using trace theory. Suppose N_E is the TPN describing the behavior of the environment, and T_E is the corresponding trace structure to N_E. The interface behavior of T_E is described by $\mathbf{del}(D)(P_E)$, where D is the set of signals internal to the environment, and P_E is the set of possible traces. Now, if the internal signals in D are removed from N_E, we define the trace structure for the abstracted environment as $T_A = \mathbf{abs}(D)(N_E)$. This function first removes signals in D from N_E, then it creates the corresponding trace structure. The interface behavior of the abstracted environment is defined as P_A. Therefore, a safe transformation must satisfy $\mathbf{del}(D)(P_E) \subseteq P_A$, where D contains the internal signals of the environment to be removed.

Transformation 1 is used when there is a single place in the preset, p_1, and postset, p_2, of a sequencing transition. This transformation requires the preset and postset of p_1 and p_2 to contain exactly one transition. This transformation removes the sequencing transition and these two places. We introduce a new place, p_3, with $l_3 = l_1 + l_2$ and $u_3 = u_1 + u_2$. We also introduce $(\bullet p_1, p_3)$ and $(p_3, p_2\bullet)$ to the flow relation. This transformation is illustrated in Figure 4.

LEMMA 3. *Transformation 1 is a safe transformation.*

Proof: Let N_E be the TPN of the environment, T_E be the trace structure of the environment before the transformation, and $T_A = \mathbf{abs}(D)(N_E)$ be the trace structure after the transformation where $D = \{\$\}$. It is obvious that $\mathbf{del}(D)(P_E) = P_A$. Therefore, transformation 1 is safe. □

Transformation 2 is depicted in Figure 5. In this case, the sequencing transition has a single place in its preset, p_1, and two places in its postset, p_2 and p_3. Again, all places must have only a single transition in its preset and postset.

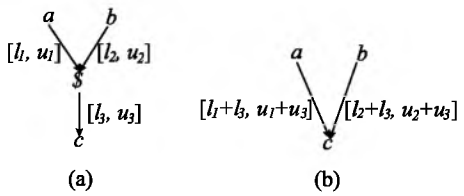

Figure 6: Safe transformation 3.

LEMMA 4. *Transformation 2 is a safe transformation.*

Proof: There are two possible untimed traces in the original net: $a\$bc$ and $a\$cb$. These map to the traces abc and acb in the abstracted net, so the first condition is satisfied. Next, we must show that the timed traces produced by the abstracted net contains all the timed traces produced by the original net with the sequencing transition deleted. Consider a timed trace $x = e_1e_2\ldots$ in which $e_i = (a, t_a)$, $e_j = (\$, t_\$)$, $e_k = (b, t_b)$, and $e_l = (c, t_c)$ with $i < j$, $j < k$, and $j < l$. The value of t_b falls in the following range:

$$t_a + l_1 + l_2 \leq t_b \leq t_a + u_1 + u_2. \tag{11}$$

The value of t_c comes from the range:

$$t_b + l_3 - u_2 \leq t_c \leq t_b + u_3 - l_2. \tag{12}$$

After abstraction, the value of t_b can still be drawn from Equation 11, but the value of t_c comes from the range:

$$t_b + (l_1 + l_3) - (u_1 + u_2) \leq t_c \leq t_b + (u_1 + u_3) - (l_1 + l_2).$$

This can be rewritten as follows:

$$t_b + (l_3 - u_2) + (l_1 - u_1) \leq t_c \leq t_b + (u_3 - l_2) + (u_1 - l_1).$$

Since $l_1 - u_1 \leq 0$ and $u_1 - l_1 \geq 0$, the range of values after abstraction is a superset of those before abstraction. This means that the abstracted net produces a superset of traces of the unabstracted net, so it is a safe transformation. □

It needs to be pointed out that this transformation creates extra interleavings between b and c not seen before the transformation. For example, after the transformation, the system could generate a trace $(a, t_a)(b, t_a + l_1 + l_2)(c, t_a + u_1 + u_3)$, where t_a is when a fires. This trace is impossible in the system before the transformation.

Figure 6 shows transformation 3 which removes a sequencing transition with two places in the preset and single place in the postset.

LEMMA 5. *Transformation 3 is a safe transformation.*

Proof: There are two possible untimed traces in the original net: $ab\$c$ and $ba\$c$. These map to the traces abc and bac in the abstracted net, so the first condition is satisfied. Next, we must show that the set of timed traces produced by the abstracted net contains all the timed traces produced by the original net with the sequencing transition deleted. Consider a timed trace $x = e_1e_2\ldots$ in which $e_i = (a, t_a)$, $e_j = (b, t_b)$, $e_k = (\$, t_\$)$, and $e_l = (c, t_c)$ with $i < k$, $j < k$, and $k < l$. The value of $t_\$$ falls in the following range:

$$\max\{t_a + l_1, t_b + l_2\} \leq t_\$ \leq \max\{t_a + u_1, t_b + u_2\} \tag{13}$$

The value of t_c comes from the range:

$$t_\$ + l_3 \leq t_c \leq t_\$ + u_3. \tag{14}$$

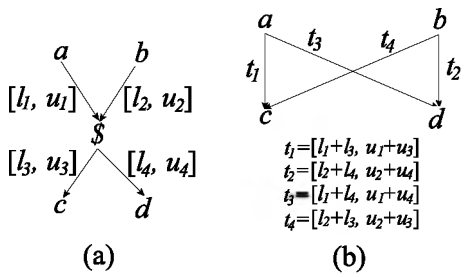

Figure 7: Safe transformation 4.

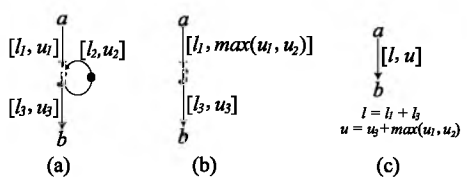

Figure 8: Safe transformation 5.

Substituting Equation 13 into Equation 14 yields:

$$\begin{aligned}&\max\{t_a + l_1, t_b + l_2\} + l_3 \leq t_c \\ &\leq \max\{t_a + u_1, t_b + u_2\} + u_3.\end{aligned} \tag{15}$$

After abstraction, the value of t_c comes from the range:

$$\begin{aligned}&\max\{t_a + l_1 + l_3, t_b + l_2 + l_3\} \leq t_c \\ &\leq \max\{t_a + u_1 + l_3, t_b + u_2 + l_3\}.\end{aligned} \tag{16}$$

This is equivalent to Equation 15, so the range of values for t_c before and after abstraction are equal. This means that the abstracted net produces the same timed traces as the unabstracted net, so it is a safe transformation. □

Transformation 2 and 3 can be naturally extended to more than two places in the postset and preset of the sequencing transition. Transformation 4 shown in Figure 7 is the combination of transformations 2 and 3. It removes a sequencing transition with two places in both its preset and postset. From above conclusions, we can prove the following lemma easily.

LEMMA 6. *Transformation 4 is a safe transformation.*

A more complicated case is when a self loop appears on a sequencing transition, as shown in Figure 8(a). Self loops must be removed before the other safe transformation can be used. This transformation changes the upper bound of the delay on each of the places in the preset of the sequencing transition to the maximum of the original upper bound and the upper bound of the place in the self loop. The lower bounds remain the same. This ensures that no matter when the last instance of the sequencing transition occurred, the self loop token would be expired when the other places in the preset become expired. This makes the self loop redundant. The new TPN is shown in Figure 8(b). After removal of the self loop, the new TPN can be transformed as described above, and the result is shown in Figure 8(c). This is transformation 5.

LEMMA 7. *Transformation 5 is a safe transformation.*

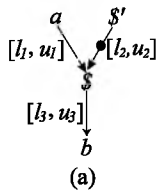

Figure 9: Unroll the self loop rule.

Proof: These nets clearly have the same untimed traces. Next, we must show that the timed traces produced by the abstracted net contains all the timed traces produced by the original net with the self loop deleted. Consider a timed trace $x = e_1 e_2 \ldots$ in which $e_i = (\$, t_\$^{-1})$, $e_j = (a, t_a)$, $e_k = (\$, t_\$)$, and $e_l = (b, t_b)$, with $i < j < k < l$. The value of $t_\$$ falls in the following range:

$$\max\{t_a + l_1, t_\$^{-1} + l_2\} \leq t_\$ \leq \max\{t_a + u_1, t_\$^{-1} + u_2\} \quad (17)$$

where $t_\$^{-1}$ represents the time of the previous \$ transition. Figure 8(a) is redrawn in Figure 9 to show this relationship where $\$'$ is the last \$ transition. After abstraction, the value of $t_\$$ falls in the following range:

$$t_a + l_1 \leq t_\$ \leq t_a + \max\{u_1, u_2\} \quad (18)$$

Since $x \leq \max\{x, y\}$ for any values of x and y, this means that $t_a + l_1 \leq \max\{t_a + l_1, t_\$^{-1} + l_2\}$. Since $t_a \geq t_\$^{-1}$, this means that $t_a + \max\{u_1, u_2\} \geq \max\{t_a + u_1, t_\$^{-1} + u_2\}$. Therefore, the range of values for $t_\$$ after abstraction is a superset of those before abstraction. This means that the abstracted net produces a superset of traces of the unabstracted net, so removing the self-loop in this way is a safe transformation. After removing the self loop, the TPN in Figure 8(b) can be processed using transformation 1, which is safe. Therefore, transformation 5 is safe. □

Our verification procedure uses the function $\mathbf{fail}(\check{P})(P)$ to check correctness of a circuit. $\check{P}$ is a predicate which takes a trace and returns whether or not the trace results in a failure. This predicate can be defined to check for properties such as hazard-freedom, minimum or maximum time separations between transitions, deadlock freedom, etc. P is a set of traces. The function $\mathbf{fail}(\check{P})(P)$ checks each property using $\check{P}$ for each trace in P, and returns a set including all traces in P that fail to satisfy $\check{P}$. For hierarchical verification to succeed, the definition $\check{P}$ must preserve the following property:

$$\mathbf{fail}(\check{P})(X) \subseteq \mathbf{fail}(\check{P})(X') \qquad \text{if } X \subseteq X' \quad (19)$$

This property states that for two sets of traces, correctness checking using the same predicate does not affect the relation of the two sets.

After safe transformations, $\mathbf{del}(D)(P_E) \subseteq P_A$ where $T_A = \mathbf{abs}(D)(N_E)$ and D contains the internal signals to be removed. This indicates that the interface behavior of the environment after transformations is a superset of that before transformations. From Equation 19, we get the following:

$$\mathbf{fail}(\check{P})(\mathbf{del}(D)(P_E)) \subseteq \mathbf{fail}(\check{P})(P_A) \quad (20)$$

This means that the failure set of the abstracted environment is a superset of the failure set of the unabstracted environment with internal signals hidden. Now we show an important lemma.

LEMMA 8. *A system is described by a TPN, N_E, its corresponding trace structure is T_E, and D contains internal signals of the system. If the function $\mathbf{abs}(D)(N_E)$ uses only safe transformations, then $\mathbf{hide}(D)(T_E) \preceq \mathbf{abs}(D)(N_E)$.*

Proof: Lemma 1. □

6. HIERARCHICAL VERIFICATION

As mentioned above, hierarchical verification picks a block in a system as the target for verification and treats the rest of ths blocks as the environment of the target block. Next, the abstraction technique is used to reduce complexity of the environment. If each block is verified individually to be failure-free with its abstracted environment, then we can prove that the entire system is failure-free. This idea is formalized in the following theorems.

Given two modules $M_1 = \langle I_1, O_1, P_1 \rangle$ and $M_2 = \langle I_2, O_2, P_2 \rangle$, we would like to verify their composition, $M_1 \parallel M_2$, is failure-free. In the following theorem, X_1 and X_2 are the internal signal sets of M_1 and M_2, respectively (i.e., $X_1 = O_1 - I_2$, $X_2 = O_2 - I_1$, and $X_1 \cap X_2 = \emptyset$).

THEOREM 1. *Let X_1 and X_2 be the internal signal sets of M_1 and M_2, respectively. If $M_1 \parallel \mathbf{hide}(X_2)(M_2)$ is failure-free, and $\mathbf{hide}(X_1)(M_1) \parallel M_2$ is failure-free, then $M = M_1 \parallel M_2$ is failure-free.*

Proof: First, the failure set of $M_1 \parallel M_2$ is

$$(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(\check{P})(P_1)) \cap \mathbf{del}^{-1}(X_1)(P_2)) \cup (\mathbf{del}^{-1}(X_1)(\mathbf{fail}(\check{P})(P_2)) \cap \mathbf{del}^{-1}(X_2)(P_1)) \quad (21)$$

Suppose $M_1 \parallel \mathbf{hide}(X_2)(M_2)$ is failure-free. This means its failure set is empty.

$$(\mathbf{fail}(\check{P})(P_1) \cap \mathbf{del}^{-1}(X_1)(\mathbf{del}(X_2)(P_2))) \cup (P_1 \cap \mathbf{del}^{-1}(X_1)(\mathbf{fail}(\check{P})(\mathbf{del}(X_2)(P_2)))) = \emptyset \quad (22)$$

Therefore,

$$\mathbf{fail}(\check{P})(P_1) \cap \mathbf{del}^{-1}(X_1)(\mathbf{del}(X_2)(P_2)) = \emptyset \quad (23)$$

$$P_1 \cap \mathbf{del}^{-1}(X_1)(\mathbf{fail}(\check{P})(\mathbf{del}(X_2)(P_2))) = \emptyset \quad (24)$$

Using Equation 5, Equation 23 can be transformed to:

$$\mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(\check{P})(P_1))) \cap \mathbf{del}^{-1}(X_1)(\mathbf{del}(X_2)(P_2)) = \emptyset \quad (25)$$

Using Equation 4, Equation 25 becomes:

$$\mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(\check{P})(P_1))) \cap \mathbf{del}(X_2)(\mathbf{del}^{-1}(X_1)(P_2)) = \emptyset \quad (26)$$

From Equation 6, Equation 26 can be transformed to:

$$\mathbf{del}(X_2)(\mathbf{del}^{-1}(X_2)(\mathbf{fail}(\check{P})(P_1)) \cap \mathbf{del}^{-1}(X_1)(P_2)) = \emptyset \quad (27)$$

Finally, from Equation 3, we get the following result:

$$\mathbf{del}^{-1}(X_2)(\mathbf{fail}(\check{P})(P_1)) \cap \mathbf{del}^{-1}(X_1)(P_2) = \emptyset \quad (28)$$

Similarly, suppose $M_2 \parallel \mathbf{hide}(X_1)(M_1)$ is failure-free. Thus, its failure set is also empty.

$$(\mathbf{fail}(\check{P})(P_2) \cap \mathbf{del}^{-1}(X_2)(\mathbf{del}(X_1)(P_1))) \cup (P_2 \cap \mathbf{del}^{-1}(X_2)(\mathbf{fail}(\check{P})(\mathbf{del}(X_1)(P_1)))) = \emptyset \quad (29)$$

$$\mathbf{fail}(\check{P})(P_2) \cap \mathbf{del}^{-1}(X_2)(\mathbf{del}(X_1)(P_1)) = \emptyset \quad (30)$$

$$P_2 \cap \mathbf{del}^{-1}(X_2)(\mathbf{fail}(\check{P})(\mathbf{del}(X_1)(P_1))) = \emptyset \quad (31)$$

By applying the same steps to Equation 30, we derive:

$$\mathbf{del}^{-1}(X_1)(\mathbf{fail}(\check{P})(P_2)) \cap \mathbf{del}^{-1}(X_2)(P_1) = \emptyset \quad (32)$$

The union of Equation 28 and 32 is the failure set of $M_1 \parallel M_2$. Since both Equation 28 and 32 are empty, the failure set of $M_1 \parallel M_2$ is empty. □

Calculation of P is an exponential problem. From Lemma 8, we know that $\mathbf{hide}(D)(E) \preceq \mathbf{abs}(D)(T)$. Therefore, combined with Lemma 2, we know $M_1 \parallel \mathbf{hide}(X_2)(M_2) \preceq M_1 \parallel \mathbf{abs}(X_2)(M_2)$ and $\mathbf{hide}(X_1)(M_1) \parallel M_2 \preceq \mathbf{abs}(X_1)(M_1) \parallel M_2$. Using the above conclusions, we show another very important theorem.

THEOREM 2. *Let X_1 and X_2 be internal signal sets of M_1 and M_2, respectively. If $M_1 \parallel \mathbf{abs}(X_2)(M_2)$ is failure-free, and $\mathbf{abs}(X_1)(M_1) \parallel M_2$ is failure-free, then $M = M_1 \parallel M_2$ is failure-free.*

Proof: Lemma 2 and Lemma 8. □

7. PRELIMINARY RESULTS

We have implemented the abstraction technique introduced above and incorporated it into our VHDL and HSE compiler [27] frontend to the ATACS tool. We have applied our abstraction method to several examples. These examples contain multiple duplicates of simple elements, and they are easy to expand. The following charts show the comparisons between the times for flat synthesis and synthesis using abstraction (similar results have been obtained for verification). All the examples have the same feature that when the number of stages grows, the state space and the synthesis time grow exponentially, and very quickly the synthesis process exits because of memory exhaustion.

The first example is a dataless version of the precharge half buffer (PCHB) from [15] (the TPN for a 2-stage version is shown in Figure 2). For this example, ATACS finishes successfully on 7 stages on the flat design; but with abstraction turned on, it easily synthesizes 100 stages in about 6.5 minutes. Comparative runtimes for the synthesis of 1 through 9 stages is shown in Figure 10.

The second example is a multiple stage controller for a self timed FIFO. In [16], a highly optimized hand designed timed circuit implementation is presented. The correctness is highly dependent on timing parameters. By using ATACS, the same efficient circuit is derived [25]. We first synthesized it using our POSET timing technique [3] without abstraction, and we can only synthesize the FIFO up to 4 stages. For the 5-stage FIFO, we had to kill the process after it ran for over a day. With abstraction, however, we can easily proceed to 100 stages, which takes approximately 23 minutes. Comparative results upto 6 stages are shown in Figure 11.

From both charts you may notice that the synthesis time with abstraction for the first few stages is a little bit larger than that by using POSETs alone. This is because abstraction contributes to part of the runtime. As the designs become more and more complex, the time for abstraction dominates the total synthesis time. However, since abstraction runtime grows polynomially in the size of the specification, the total synthesis time with abstraction grows in

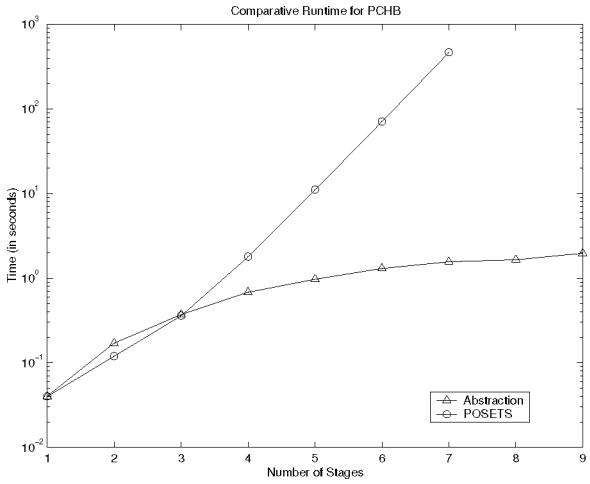

Figure 10: Synthesis time for PCHB example.

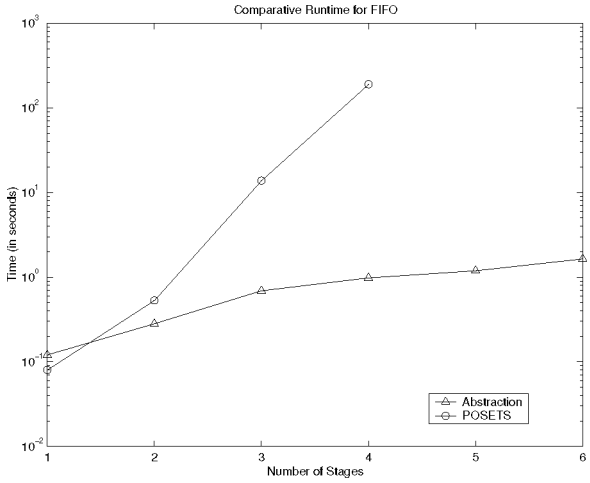

Figure 11: Synthesis time for FIFO example.

an approximately polynomial manner. This is substantially better than the exponential growth in the analysis of flat designs. We have also found that synthesis with abstraction is not only several orders of magnitude faster than flat synthesis, but also successful on several orders of magnitude more complex designs than flat synthesis.

8. CONCLUSIONS

State space explosion is the bottleneck of current synthesis and verification methods. This paper presents a new approach to avoid it by partitioning the designs into manageable blocks and considering the blocks one by one. By using the hierarchy information provided in the specification, the environment is simplified, and the state space for each block is reduced by orders of magnitude. Experimental results show that synthesis and verification with abstraction is not only several orders of magnitude faster but also capable of analyzing systems several orders of magnitude more complex than flat analysis. Currently, our technique is unable to abstract transitions involved in choices, so we plan in the future to extend our technique to general TPNs.

9. REFERENCES

[1] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng. Checking combinational equivalence of speed-independent circuits. *Formal Methods in System Design*, Mar. 1998.

[2] P. A. Beerel, T. H.-Y. Meng, and J. Burch. Efficient verification of determinate speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 261–267. IEEE Computer Society Press, Nov. 1993.

[3] W. Belluomini and C. Myers. Verification of timed systems using posets. In *International Conference on Computer Aided Verification*. Springer-Verlag, 1998.

[4] W. Belluomini and C. J. Myers. Timed event-level structures. In *Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, Austin, Texas, USA, Dec. 1997.

[5] W. Belluomini, C. J. Myers, and H. P. Hofstee. Verification of delayed-reset domino circuits using ATACS. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, Apr. 1999.

[6] G. Berthelot. Checking properties of nets using transformations. In *Lecture Notes in Computer Science, 222*, pages 19–40, 1986.

[7] R. K. Brayton. Vis: A system for verification and synthesis. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 428–432, 1996.

[8] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.

[9] Y. H. D. Moundanos, J. Abraham. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers.*, 47(1):2–14, 1998.

[10] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[11] J. Gu and R. Puri. Asynchronous circuit synthesis with boolean satisfiability. In *IEEE Trans. CAD, Vol. 14, No. 8*, pages 961–973, 1995.

[12] R. Johnsonbaugh and T. Murata. Additional methods for reduction and expansion of marked graphs. In *IEEE Trans. Ciruits Systems, vol. CAS-28, no. 1*, pages 1009–1014, 1981.

[13] S. T. Jung and C. J. Myers. Direct synthesis of timed asynchronous circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 332–337, Nov. 1999.

[14] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[15] A. J. Martin, A. Lines, R. Manohar, and M. Nystrom. The design of an asynchronous mips r3000 microprocessor. In R. B. Brown and A. T. Ishii, editors, *1997 Michigan Conference on Very Large Scale Integration*, pages 164–181, 1997.

[16] C. E. Molnar, I. W. Jones, B. Coates, and J. Lexau. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289. IEEE Computer Society Press, Apr. 1997.

[17] T. Murata. Petri nets: Properties, analysis, and applications. In *Proceedings of the IEEE 77(4)*, pages 541–580, 1989.

[18] T. Murata and J. Y. Koh. Reduction and expansion of lived and safe marked graphs. In *IEEE Trans. Ciruits Systems, vol. CAS-27, no. 10*, pages 68–70, 1980.

[19] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Massachusetts Inst. of Tech., Feb. 1974.

[20] R. A. R.Grosu and M. McDougall. Efficient reachability analysis of hierarchical reactive machines, 1999. Submitted.

[21] O. Roig. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Univsitat Politècnia de Catalunya, May 1997.

[22] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, Apr. 1999.

[23] I. Suzuki and T. Murata. *Stepwise refinements for transitions and places*. New York: Springer-Verlag, 1982.

[24] I. Suzuki and T. Murata. A method for stepwise refinements and abstractions of petri nets. In *Journal Of Computer System Science, 27(1)*, pages 51–76, 1983.

[25] R. A. Thacker, W. Belluomini, and C. J. Myers. Timed circuit synthesis using implicit methods. In *Proc. International Conference on VLSI Design*, pages 181–188, 1999.

[26] T. Yoneda and H. Ryu. Timed trace theoretic verification using partial order reduction. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–121, Apr. 1999.

[27] H. Zheng. Specification and compilation of timed systems. Master's thesis, University of Utah, 1998.